# secmodel_sandbox

## An application sandbox for NetBSD

Stephen Herwig

# sandboxing

Sandboxing: limiting the privileges of a process

Two motivations

- Running untrusted code in a restricted environment
- Dropping privileges in trusted code so as to reduce the attack surface in the event of an unforeseen vulnerability

# many os-level implementations

- systrace
- SELinux
- AppArmor
- seccomp(-bpf)
- Apple's Sandbox (formerly Seatbelt)
- Capsicum
- OpenBSD's pledge syscall

Rich design space:

- which use cases are supported?
- footprint (system-wide or process-wide)
- are policies embedded in program or external?
- when are policies loaded?
- expressiveness of policies?
- portability

# secmodel_sandbox high-level design

- Implemented as a kernel module
- Sandbox policies are Lua scripts
- A process sets the policy script via an ioctl
- The kernel evaluates the script using NetBSD's experimental in-kernel Lua interpreter
- The output of the evaluation are rules that are attached to the process's credential and checked during privileged authorization requests

# secmodel_sandbox properties

- Sandboxes are inherited during fork and preserved over exec
- Processes may apply multiple policies: the sandbox is the union of all policies
- Policies can only further restrict privileges
- Rules may be boolean or Lua functions (functional rules)
- Functional rules may be stateful and may dynamically create new rules or modify existing rules

# secmodel_sandbox properties

- Sandboxes are inherited during fork and preserved over exec
- Processes may apply multiple policies: the sandbox is the union of all policies
- Policies can only further restrict privileges

- Rules may be boolean or Lua functions (functional rules)
- Functional rules may be stateful and may dynamically create new rules or modify existing rules

# sandbox policies: blacklist

**Policy**

```
sandbox.default('allow');

-- no forking
sandbox.deny('system.fork')

-- no networking
sandbox.deny('network')

-- no writing to files
sandbox.deny('vnode.write_data')
sandbox.deny('vnode.append_data')

-- no changing file metadata
sandbox.deny('vnode.write_times')
sandbox.deny('vnode.change_ownership')
sandbox.deny('vnode.write_security')
```

**Program**

```
main()
{
 /* initialize */
   . . .

 sandbox(POLICY);

 /* process loop */
 . . .

 return (0);
}
```

# sandbox policies: functional rules

```
sandbox.default('deny')
-- allow reading files
sandbox.allow('vnode.read_data')
-- only allow writes in /tmp
sandbox.on('vnode.write_data',
  function(req, cred, f)
    if string.find(f.name, '/tmp/') == 1 then
      return true
    else
      return false
    end
  end)
-- only allow unix domain sockets
sandbox.on('network.socket.open',
  function(req, cred, domain, typ, proto)
    if domain == sandbox.AF_UNIX then
      return true
    else
      return false
    end
  end)
```

# sandbox-exec

```
int
main(int argc, char *argv[])
{
    sandbox_from_file(argv[0]);
    execv(argv[1], &argv[1]);
    return (0);
}
```

```
$ sandbox-exec no-network.lua /usr/pkg/bin/bash
$ wget http://www.cs.umd.edu/
wget: unable to resolve host address ‘www.cs.umd.edu'
```

# kauth

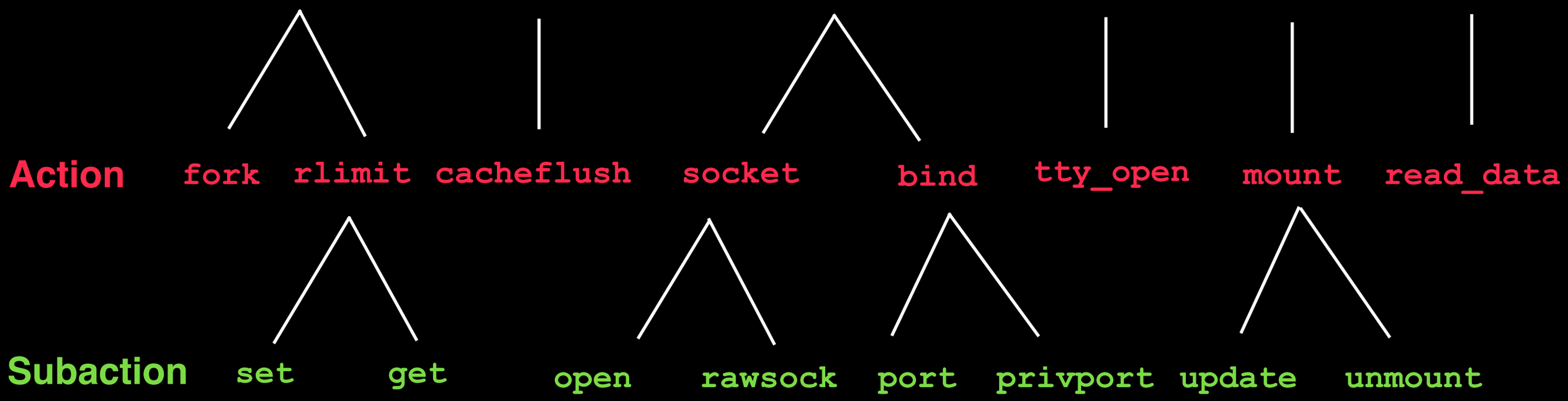
- kernel subsystem that handles all authorization requests within the kernel

- clean room implementation of subsystem in macOS

- separates security policy from mechanism

# kauth requests

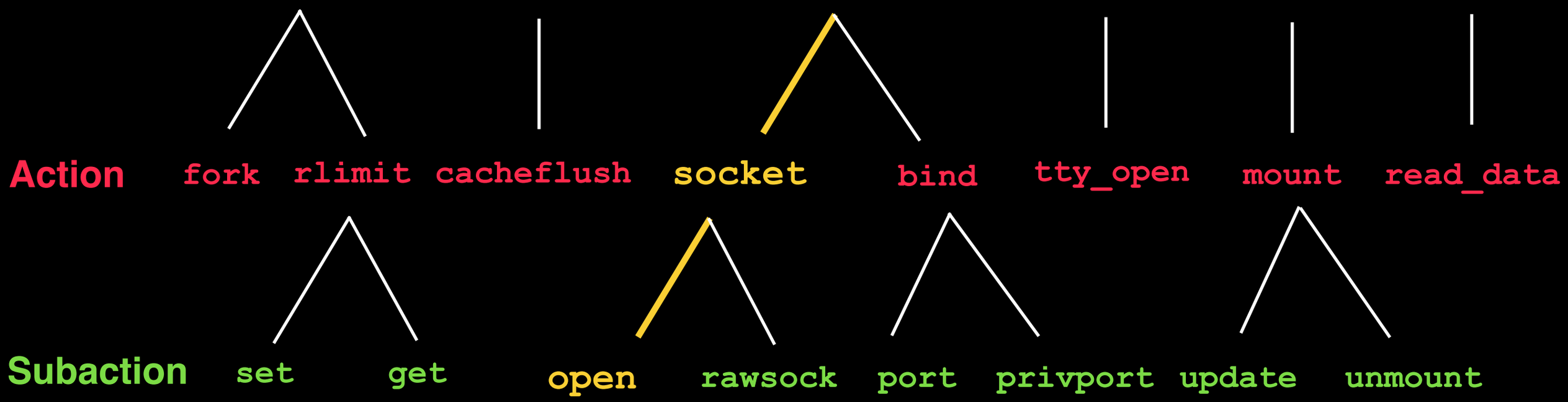
```
request := (scope, action [, subaction])
```



# kauth requests

request := (scope, action [, subaction])

Example:
creating a socket => (network, socket, open)



# kauth request to syscall mapping

Some kauth requests map directly to a syscall:

```
system.mknod  =>  mknod
```

Some kauth requests map to multiple syscalls:

```
process.setsid => {setgroups setlogin setuid
  setuid setreuid setgid setegid setregid}
```

Some syscalls trigger one of several kauth requests, depending on the syscall arguments:

```
mount(MNT_GETARGS) => system.mount.get
mount(MNT_UPDATE)  => system.mount.update
```

Many syscalls do not trigger a kauth request at all:

```
accept close dup execve flock getdents getlogin
getpeername getpid getrlimit getsockname . . .
```

# kauth request flow

**kauth uses an observer pattern.**

```
syscall(arg1, …, argn)
```

user space

kernel space

## kauth listener #1

```
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);

int cb(cred, op, ctx) {
  . . .
  return (KAUTH_RESULT_ALLOW);
}
```

## kauth listener #2

```
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);

int cb(cred, op, ctx) {
  . . .
  return (KAUTH_RESULT_ALLOW);
}
```

## syscall handler

```
kauth_authorize_action(cred, req, ctx);
```

## kauth

```
foreach (listener in scope) {
  error = listener->cb(cred, op, ctx);
  if (error == KAUTH_RESULT_ALLOW)
    allow = 1;
  else if (error == KAUTH_RESULT_DENY)
    fail = 1;
}
if (fail) return (EPERM);
if (allow) return (0);
return (EPERM);
```

**list of network scope listeners**

```
[ lists for other scope listeners ]
```

# kauth request flow

**Subsystems interested in kauth requests register with kauth via `kauth_listen_scope()`.**

`syscall(arg1, …, argn)`

user space

kernel space

## kauth listener #1

```
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);

int cb(cred, op, ctx) {
  . . .
  return (KAUTH_RESULT_ALLOW);
}
```

## kauth listener #2

```
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);

int cb(cred, op, ctx) {
  . . .
  return (KAUTH_RESULT_ALLOW);
}
```

## syscall handler

```
kauth_authorize_action(cred, req, ctx);
```

## kauth

```
foreach (listener in scope) {
  error = listener->cb(cred, op, ctx);
  if (error == KAUTH_RESULT_ALLOW)
    allow = 1;
  else if (error == KAUTH_RESULT_DENY)
    fail = 1;
}
if (fail) return (EPERM);
if (allow) return (0);
return (EPERM);
```

list of network scope listeners

[ lists for other scope listeners ]

# kauth request flow

**Most syscalls issue an authorization request in their corresponding handler via `kauth_authorize_action()`.**

`syscall(arg1, …, argn)`

user space

kernel space

## kauth listener #1

```
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);

int cb(cred, op, ctx) {
  . . .
  return (KAUTH_RESULT_ALLOW);
}
```

## kauth listener #2

```
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);

int cb(cred, op, ctx) {
  . . .
  return (KAUTH_RESULT_ALLOW);
}
```

## syscall handler

```
kauth_authorize_action(cred, req, ctx);
```

## kauth

```
foreach (listener in scope) {
  error = listener->cb(cred, op, ctx);
  if (error == KAUTH_RESULT_ALLOW)
    allow = 1;
  else if (error == KAUTH_RESULT_DENY)
    fail = 1;
}
if (fail) return (EPERM);
if (allow) return (0);
return (EPERM);
```

list of network scope listeners

[ lists for other scope listeners ]

# kauth request flow

**`kauth_authorize_action()` iterates through each listener for the given scope, calling that listener's callback.**

`syscall(arg1, …, argn)`

user space

kernel space

## kauth listener #1

```
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);

int cb(cred, op, ctx) {
   . . .
   return (KAUTH_RESULT_ALLOW);
}
```

## kauth listener #2

```
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);

int cb(cred, op, ctx) {
   . . .
   return (KAUTH_RESULT_ALLOW);
}
```

## syscall handler

```
kauth_authorize_action(cred, req, ctx);
```

## kauth

```
foreach (listener in scope) {
   error = listener->cb(cred, op, ctx);
   if (error == KAUTH_RESULT_ALLOW)
      allow = 1;
   else if (error == KAUTH_RESULT_DENY)
      fail = 1;
}
if (fail) return (EPERM);
if (allow) return (0);
return (EPERM);
```

list of network scope listeners

[ lists for other scope listeners ]

# kauth request flow

**Generally, if any listener returns DENY, the request is denied; if any returns ALLOW and none returns DENY, the request is allowed; otherwise, the request is denied.**

```
syscall(arg1, …, argn)
```

user space

kernel space

## kauth listener #1

```
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);

int cb(cred, op, ctx) {
  . . .
  return (KAUTH_RESULT_ALLOW);
}
```

## kauth listener #2

```
kauth_listen_scope(KAUTH_SCOPE_NETWORK, cb);

int cb(cred, op, ctx) {
  . . .
  return (KAUTH_RESULT_ALLOW);
}
```

## syscall handler

```
kauth_authorize_action(cred, req, ctx);
```

## kauth

```
foreach (listener in scope) {
  error = listener->cb(cred, op, ctx);
  if (error == KAUTH_RESULT_ALLOW)
    allow = 1;
  else if (error == KAUTH_RESULT_DENY)
    fail = 1;
}
if (fail) return (EPERM);
if (allow) return (0);
return (EPERM);
```

list of network scope listeners

```
[ lists for other scope listeners ]
```

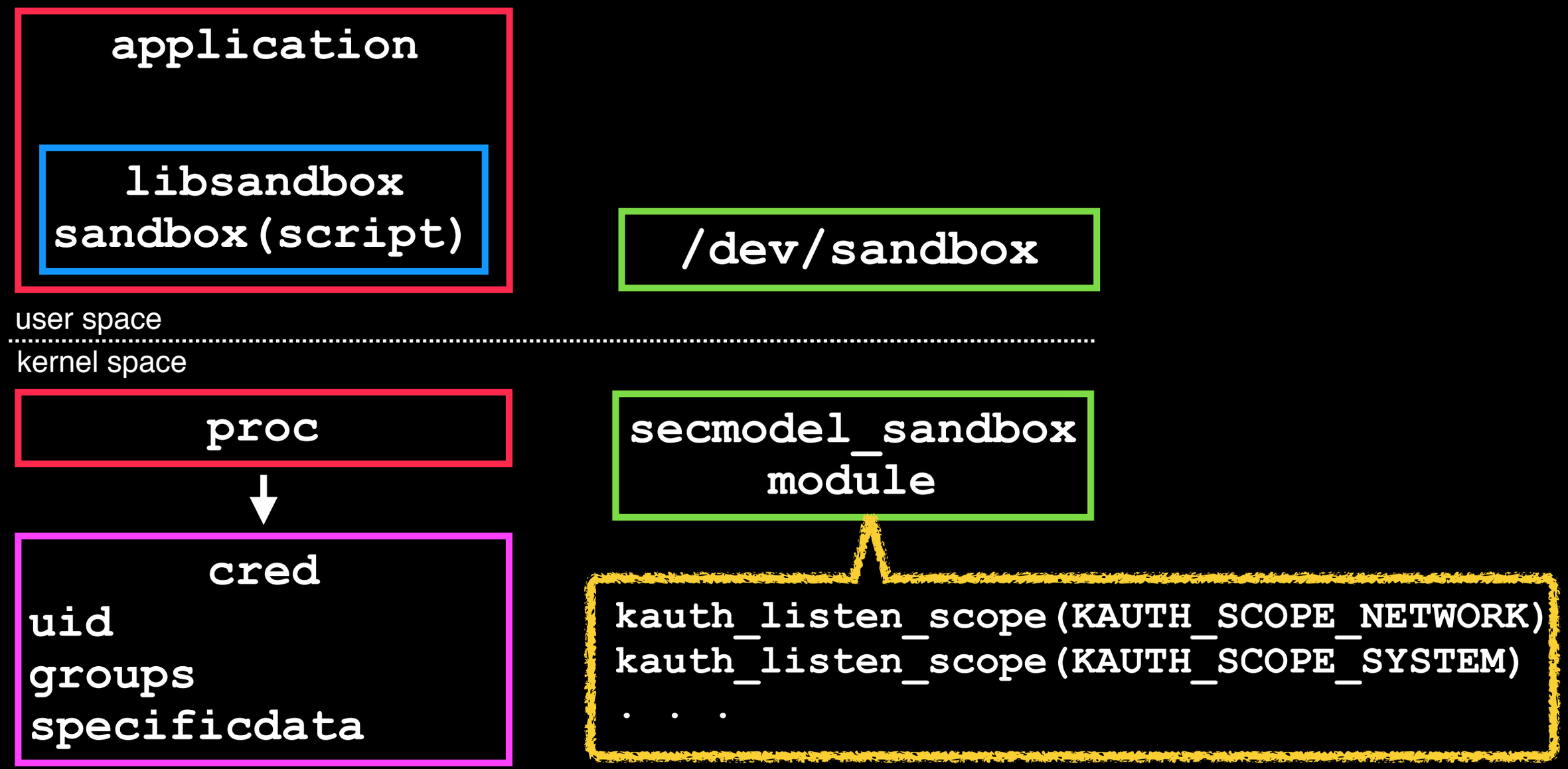
# secmodel

A security model (secmodel) is a small framework for managing a set of related kauth listeners. Fundamentally, it presents a template pattern:

```
static kauth_listener_t l_system, l_network, . . .;

void
secmodel_foo_start(void)
{
  l_system = kauth_listen_scope(KAUTH_SCOPE_SYSTEM, secmodel_foo_system_cb, NULL);
  l_network = kauth_listen_scope(KAUTH_SCOPE_NETWORK, secmodel_foo_network_cb, NULL);
  . . .
}

void
secmodel_foo_stop(void)
{
  kauth_unlisten_scope(l_system);
  kauth_unlisten_scope(l_network);
  . . .
}
```

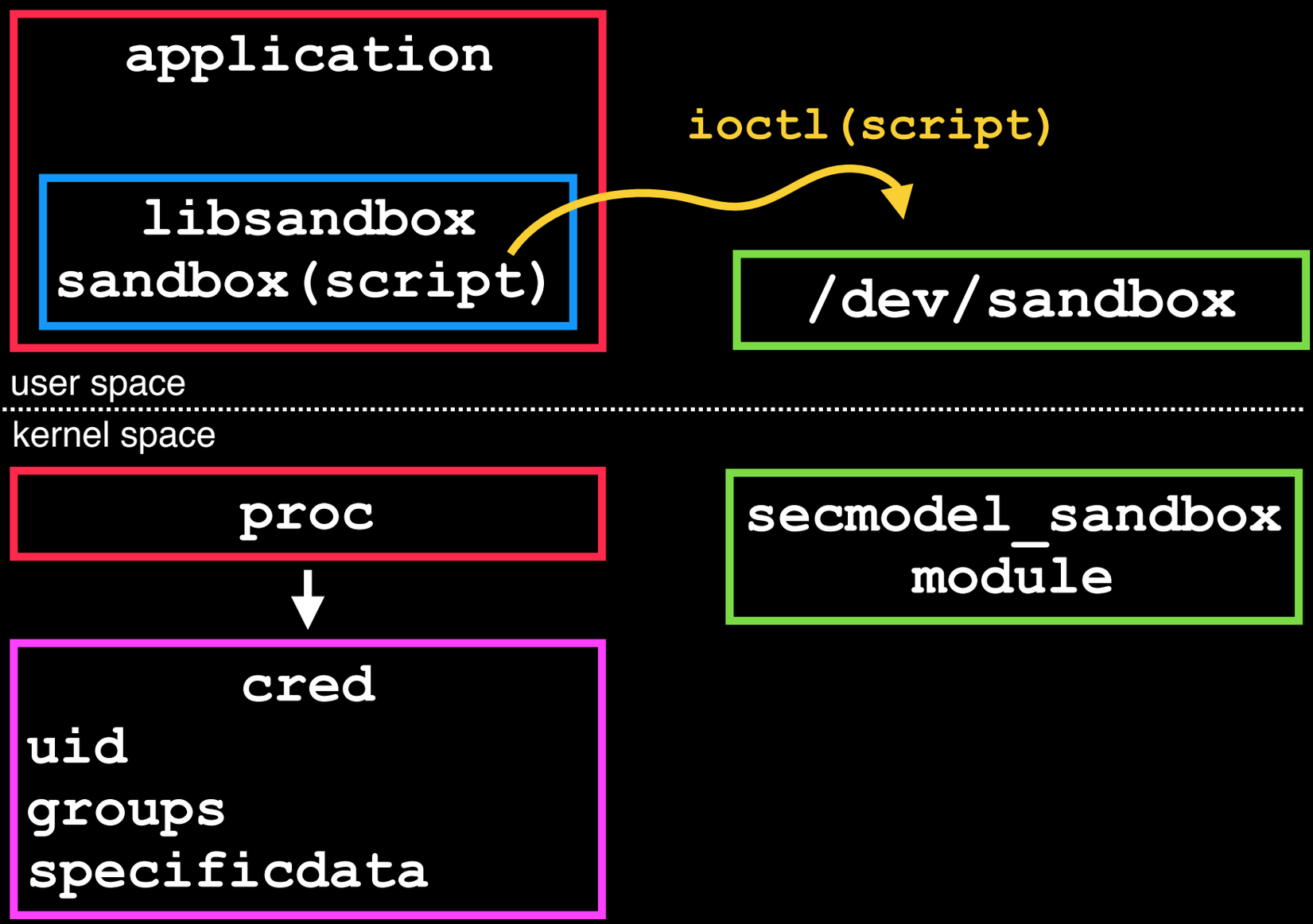
# secmodel_sandbox design

**The sandbox module registers listeners for all kauth scopes.**



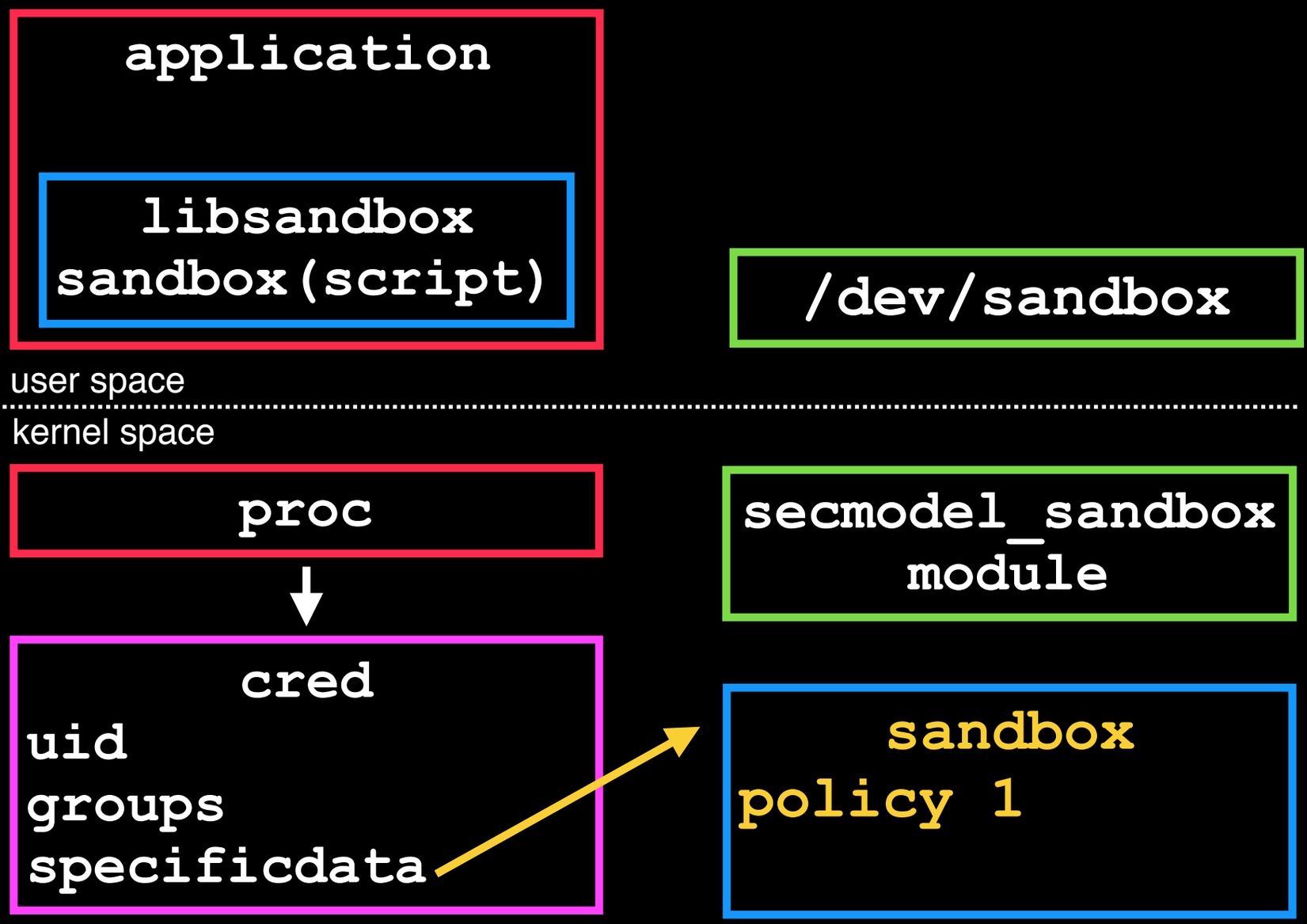
# secmodel_sandbox design

**Applications link to libsandbox. Calls to sandbox() issue an ioctl to `/dev/sandbox`, specifying the policy script.**



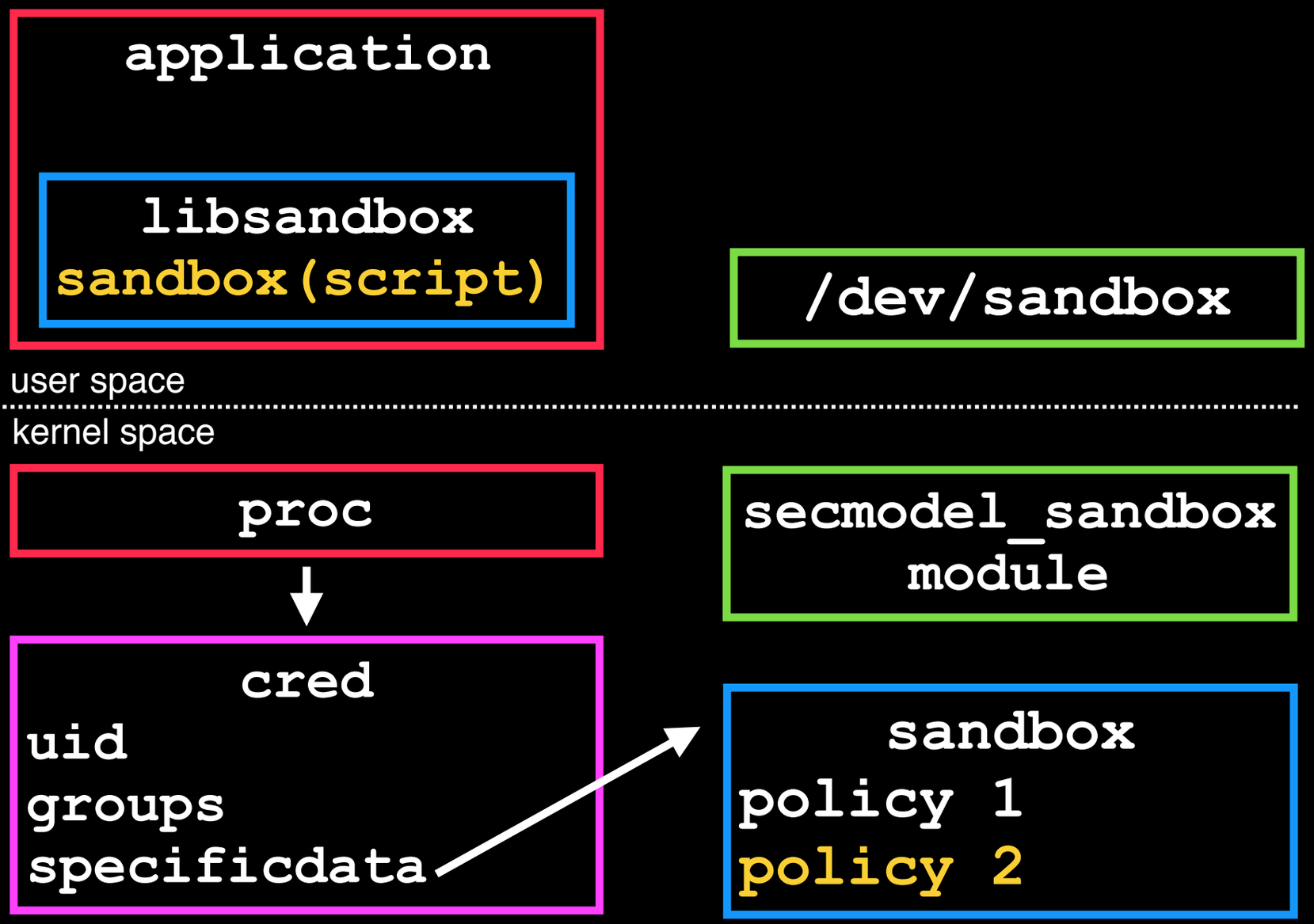
# secmodel_sandbox design

**The sandbox module services the ioctl, creates a sandbox, initializes the sandbox with the script's policy rules, and attaches the sandbox to the process's cred.**



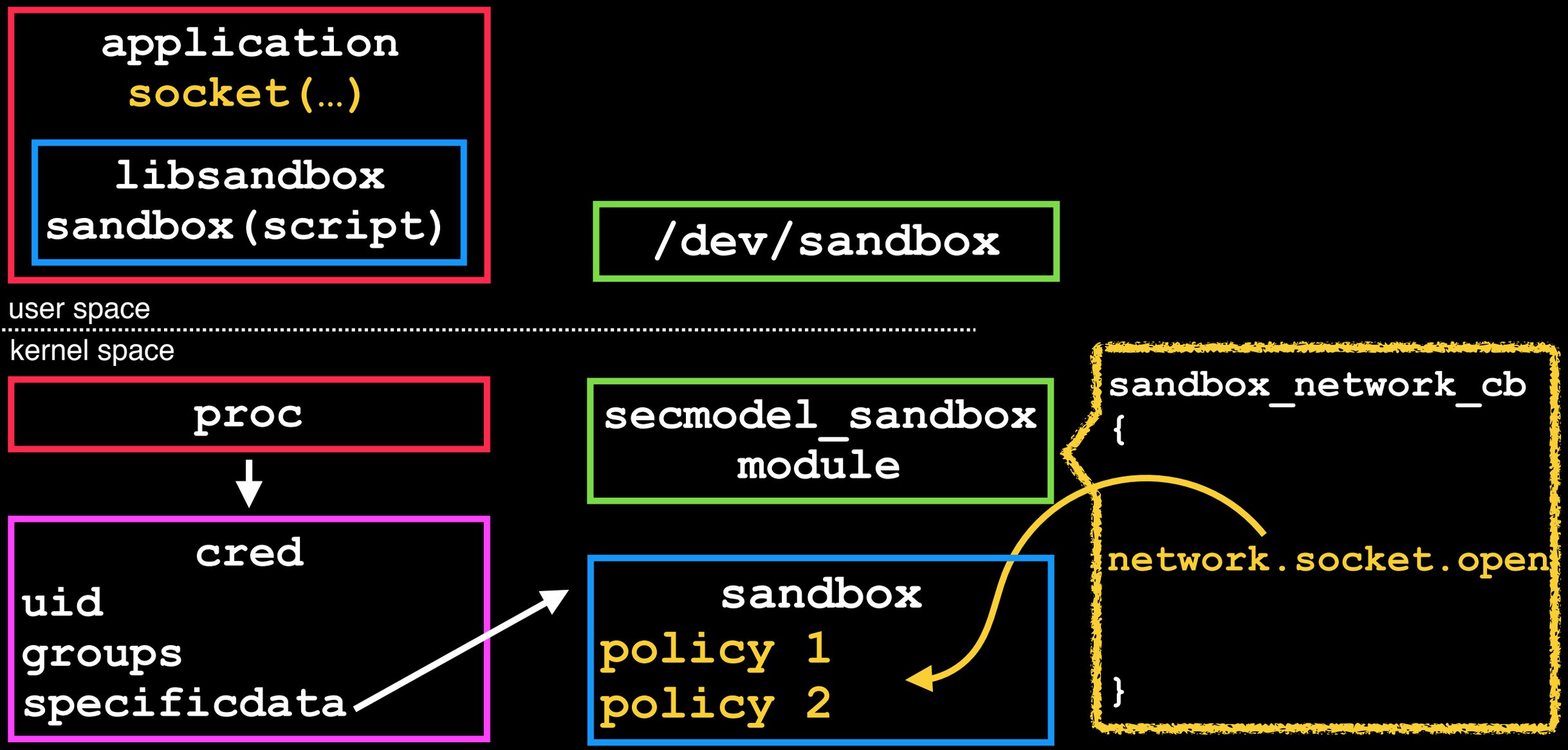
# secmodel_sandbox design

**Subsequent calls to `sandbox()` add new policies. The sandbox is collectively the union of all of its policies.**



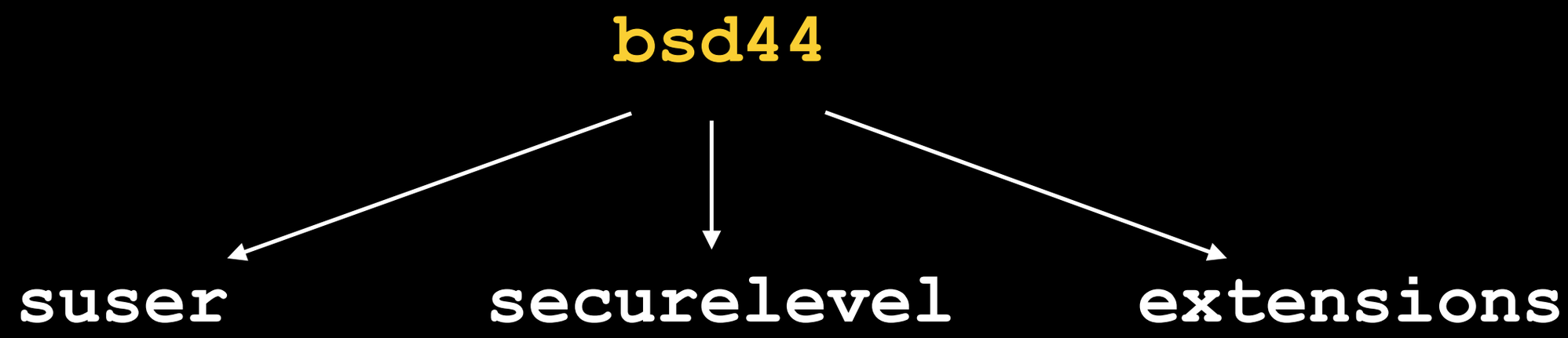
# secmodel_sandbox design

**When a syscall emits a kauth request, the secmodel_sandbox's listener checks if the process's cred has a sandbox; if so, it evaluates the request against all policies.**



# stock secmodels

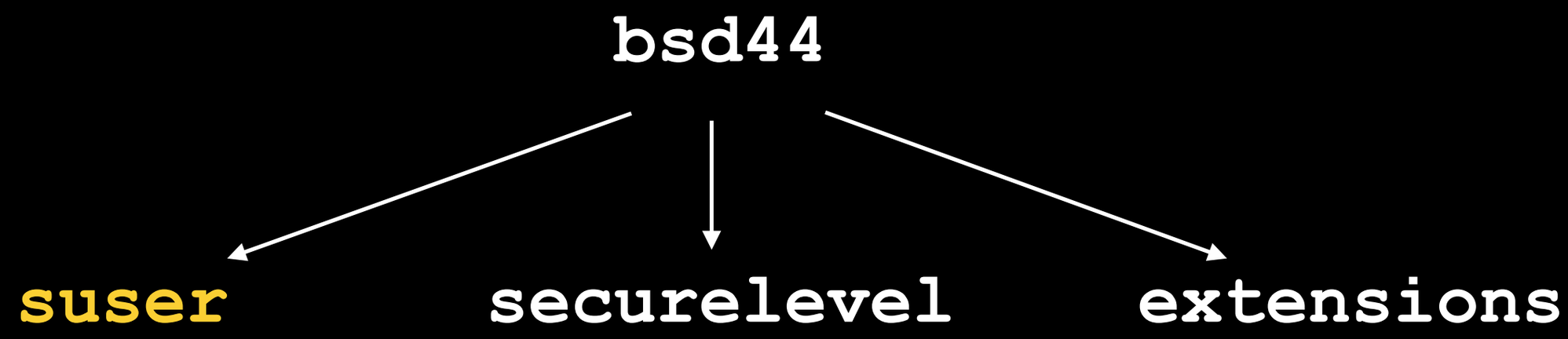
**bsd44** is the default security model, and is composed of three separate models: suser, securelevel, and extensions.



# stock secmodels

**suser** implements the traditional root user as the user with effective-id 0.

Each listener is a **whitelist**: if the requesting cred is root, then the listeners return `KAUTH_RESULT_ALLOW`; otherwise, `KAUTH_RESULT_DEFER`.



# stock secmodels

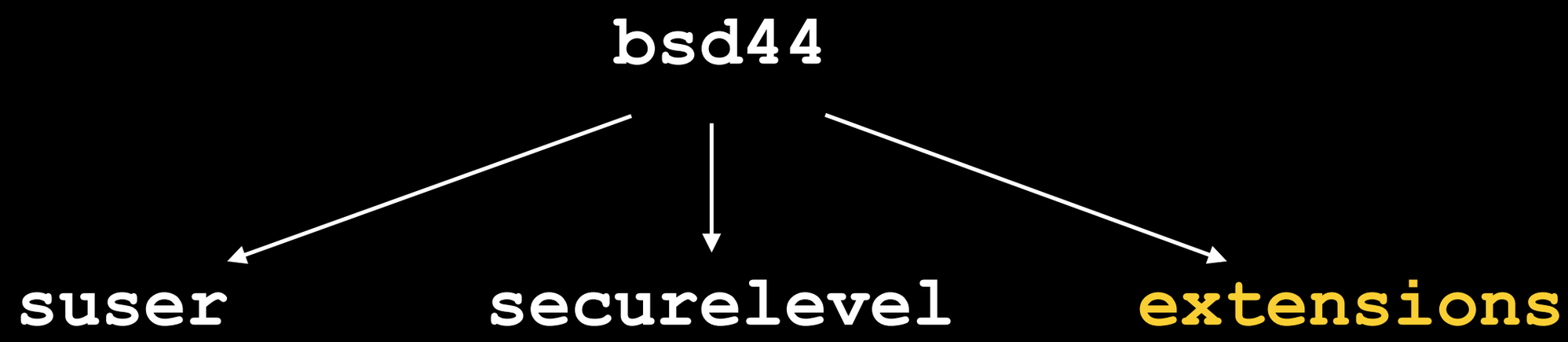
**securelevel** is a system-global policy that restricts certain operations for all users, including root.

Each listener is a **blacklist**: request decisions default to `KAUTH_RESULT_DEFER` unless explicitly forbidden, in which case the model returns `KAUTH_RESULT_DENY`.

# stock secmodels

**extensions** grant additional privileges to ordinary users, such as user-mounts and user control of CPU sets, or enable isolation measures, such as curtain mode.

extensions is implemented as a **mix of blacklists and whitelists**.



# defer revisited

**While all listeners returning DEFER usually results in a DENIED request, for the vnode scope, the last resort decision is based on traditional BSD 4.4 file access permissions.**

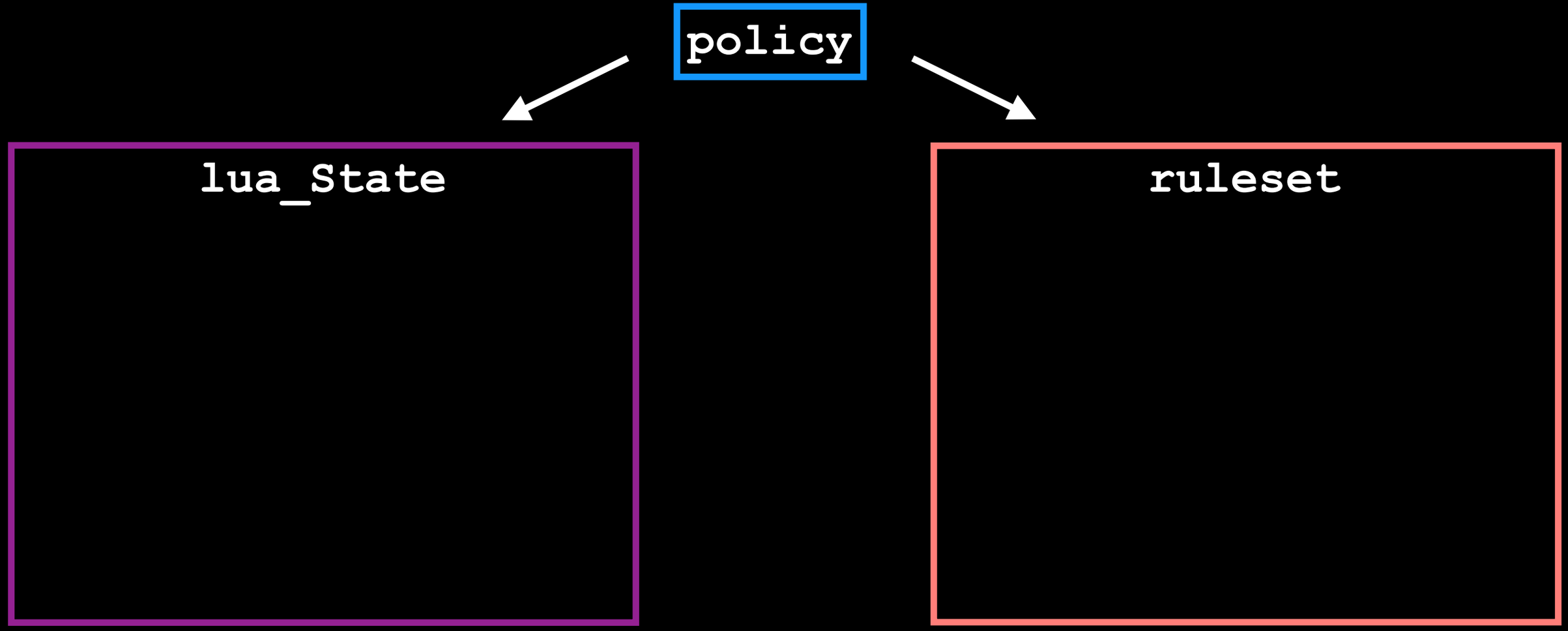
**In order to not allow elevation of privileges, secmodel_sandbox converts sandbox policy decisions of ALLOW to DEFER.**

```
-- if not internally converted to DEFER, would allow
-- reading any file
sandbox.allow('vnode.read_data')

-- if not internally converted to DEFER, would allow
-- user to load and unload modules
sandbox.allow('system.module')
```

# sandbox implementation

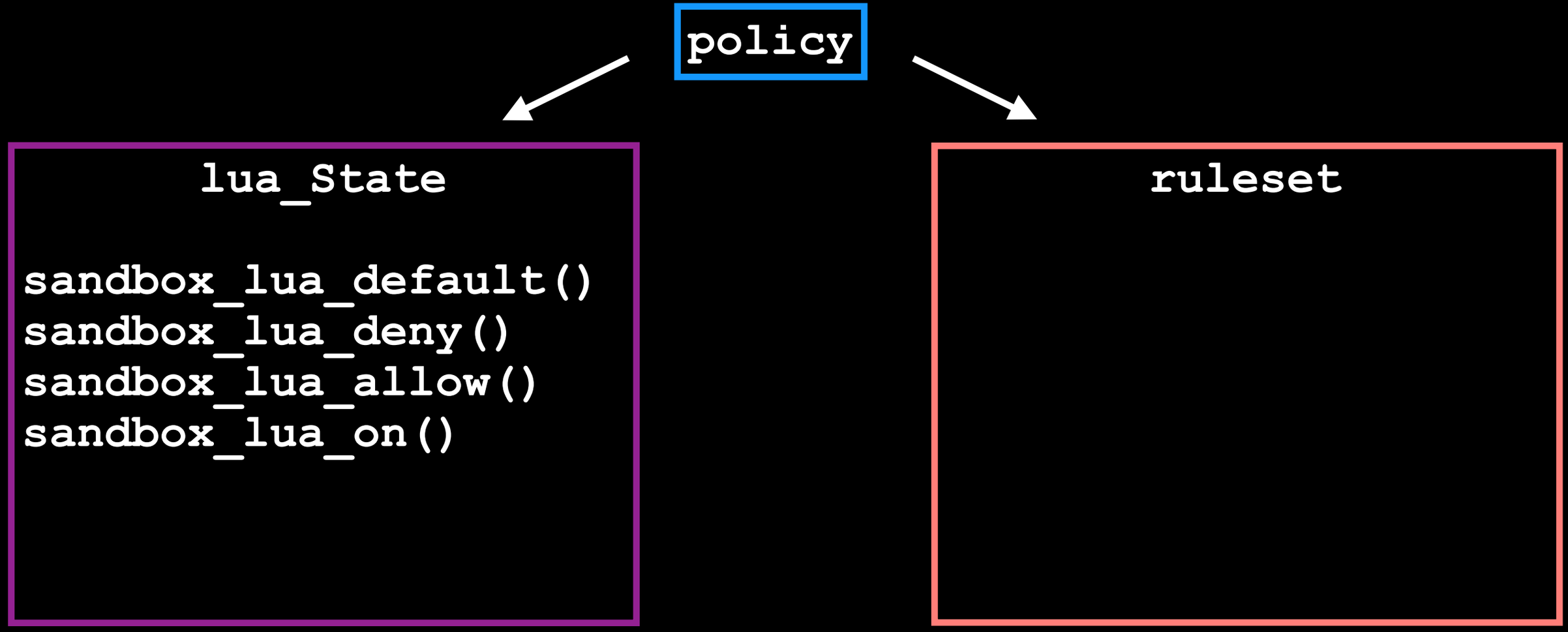
In the kernel, a policy has two main items:

- a Lua state (Lua virtual machine)
- ruleset



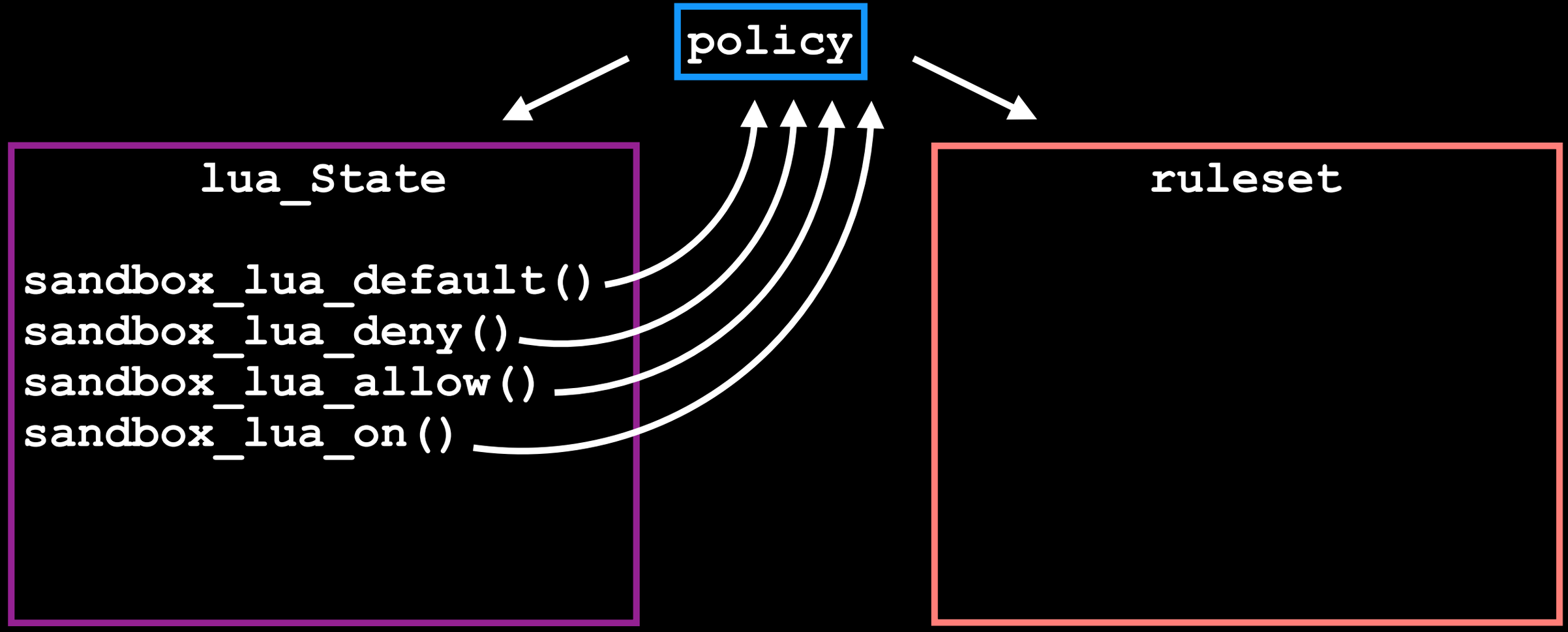
# sandbox implementation

**Before secmodel_sandbox evaluates the Lua script in the `lua_State`, secmodel_sandbox populates the `lua_State` with the `sandbox` functions and constants.**



# sandbox implementation

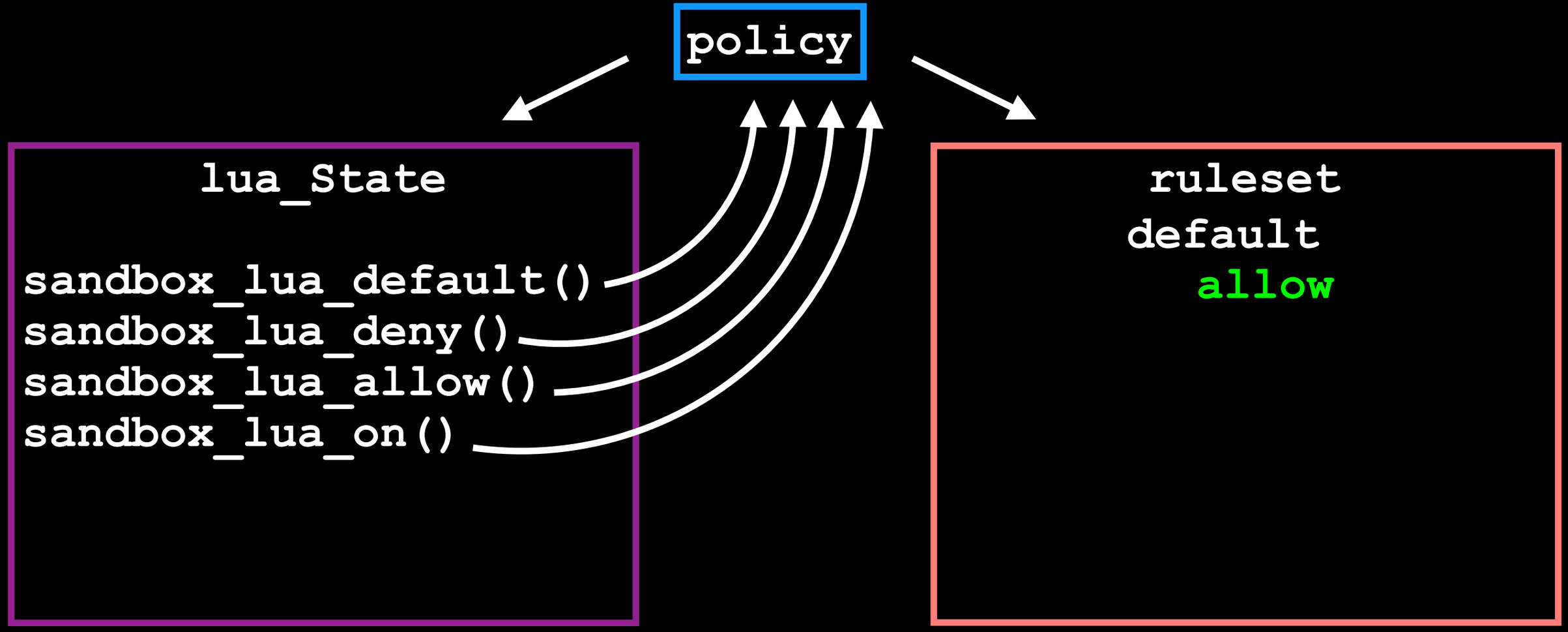
Each `sandbox` Lua function is a closure that contains a pointer back to the `policy`. In Lua terminology, the `policy` is a light userdata upvalue.



# sandbox implementation

When a `sandbox.default()`, `sandbox.allow()`, or `sandbox.deny()` function is evaluated in a script, the corresponding C function accesses the `ruleset` from the `policy` upvalue, and stores the decision for that rule.

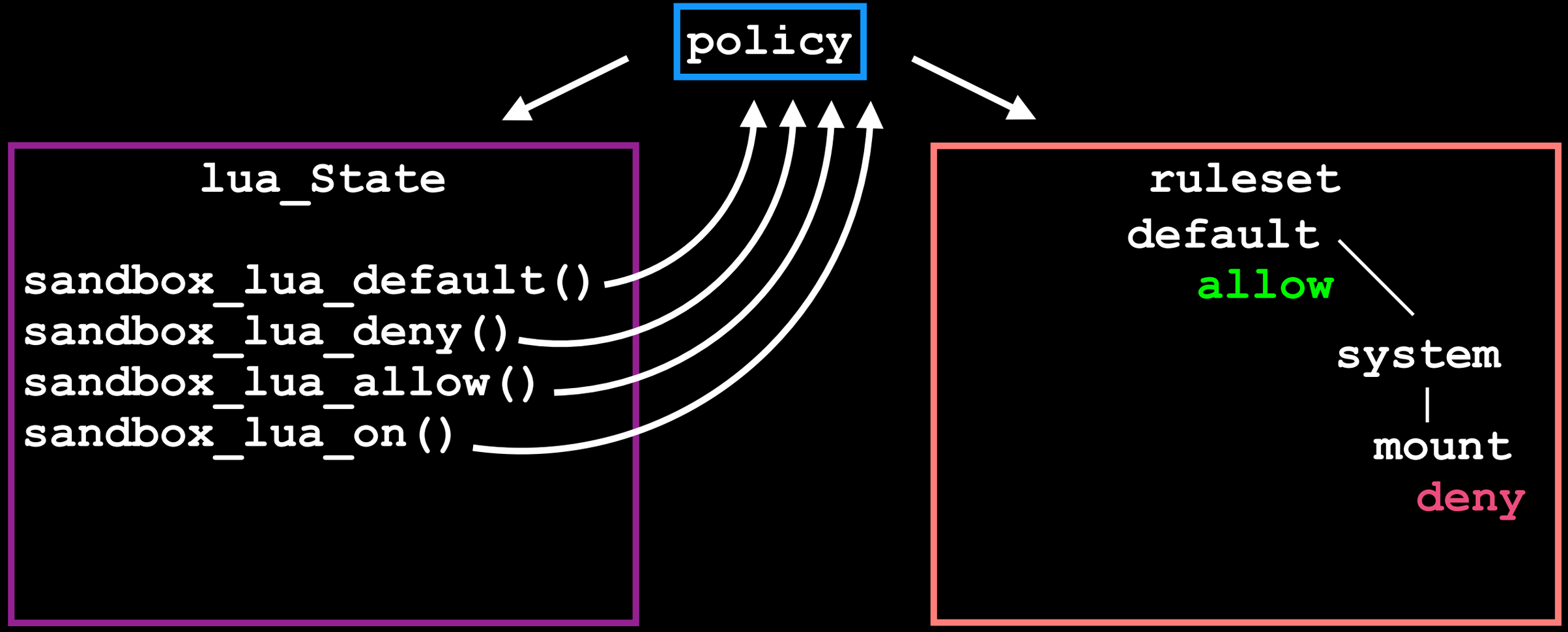
```
sandbox.default('allow')
```



# sandbox implementation

When a `sandbox.default()`, `sandbox.allow()`, or `sandbox.deny()` function is evaluated in a script, the corresponding C function accesses the `ruleset` from the `policy` upvalue, and stores the decision for that rule.

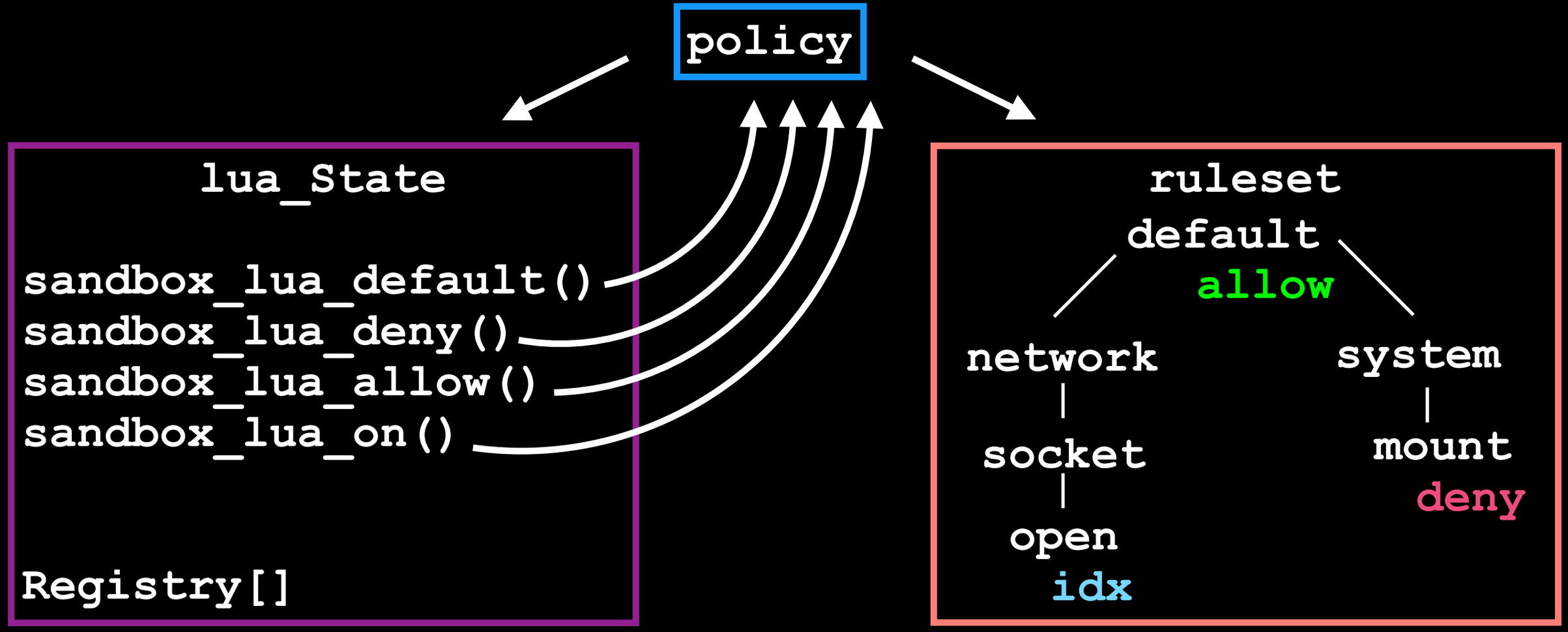
```
sandbox.deny('system.mount')
```



# sandbox implementation

**When a `sandbox.on()` rule is evaluated, the corresponding C function stores the Lua callback function for the rule in the Lua Registry.**

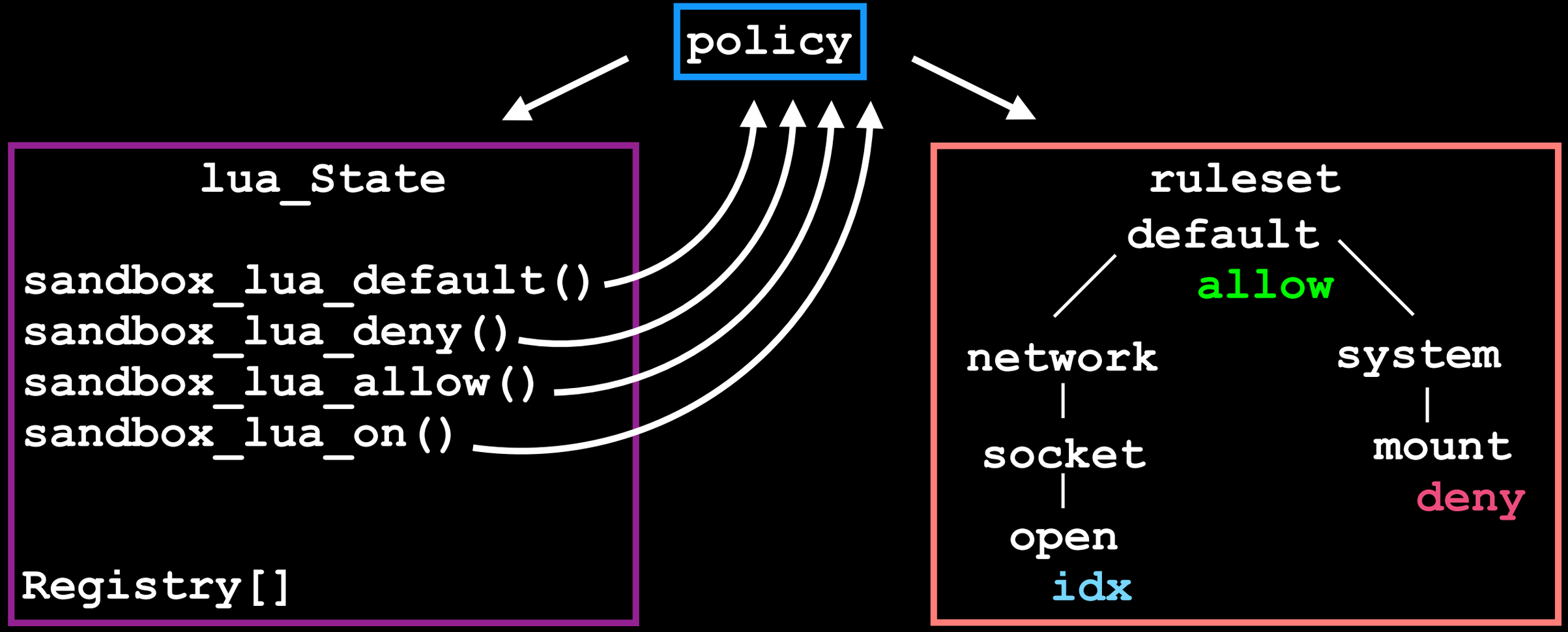
```
sandbox.on('network.socket.open',function() … end)
```



# sandbox implementation

**During a kauth request, secmodel_sandbox looks in the `ruleset` for the best matching rule.**

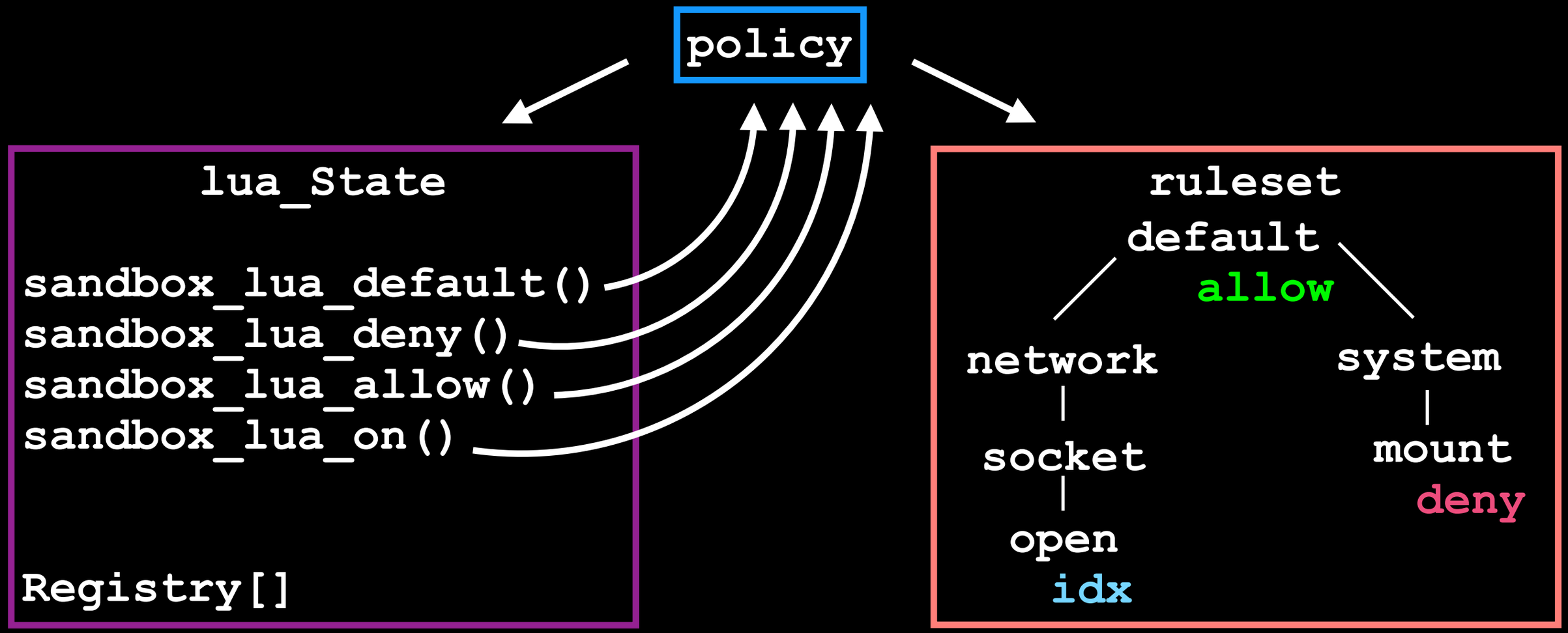
```
request:   (system, time, adjtime)
matches:   default rule
decision:  allow
```



# sandbox implementation

During a kauth request, secmodel_sandbox looks in the `ruleset` for the best matching rule.

```
request:   (system, mount, update)
matches:   system.mount rule
decision:  deny
```



# multiple policies

A process's sandbox may have multiple policies.

policies are isolated; each has it's own `lua_State` and `ruleset`.

During a kauth request for a process, each policy is evaluated.

In effect, a sandbox is a per-process kauth listener.

# multiple sandboxes

## Policy_1

```
_ = sandbox
_.default('deny')
_.allow('vnode.read')
-- needed for sandbox() ioctl
_.allow(
   'device.rawio_spec.rw'
)
```

## Policy_2

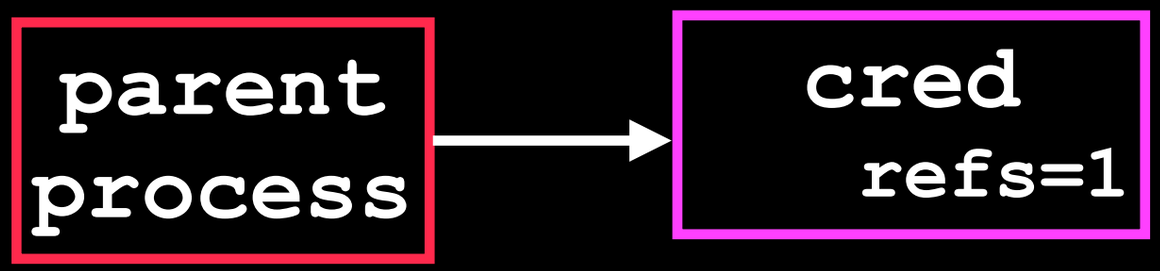
```
sandbox.default('deny')
```

## Program

```
main()
{
   sandbox(POLICY_1);

   read_file("input.dat")

   sandbox(POLICY_2)

   /* pure computation */



}
```

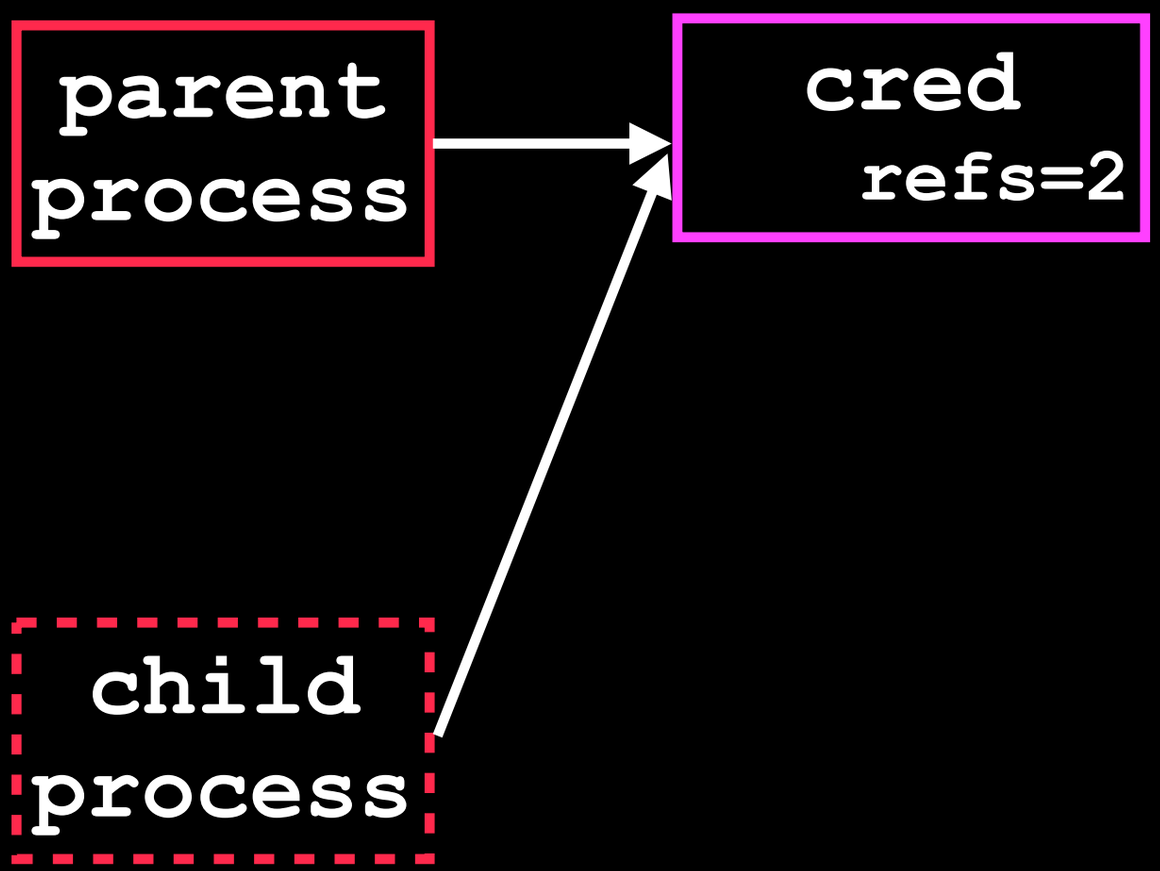
# process forking

A process contains a pointer to a credential.



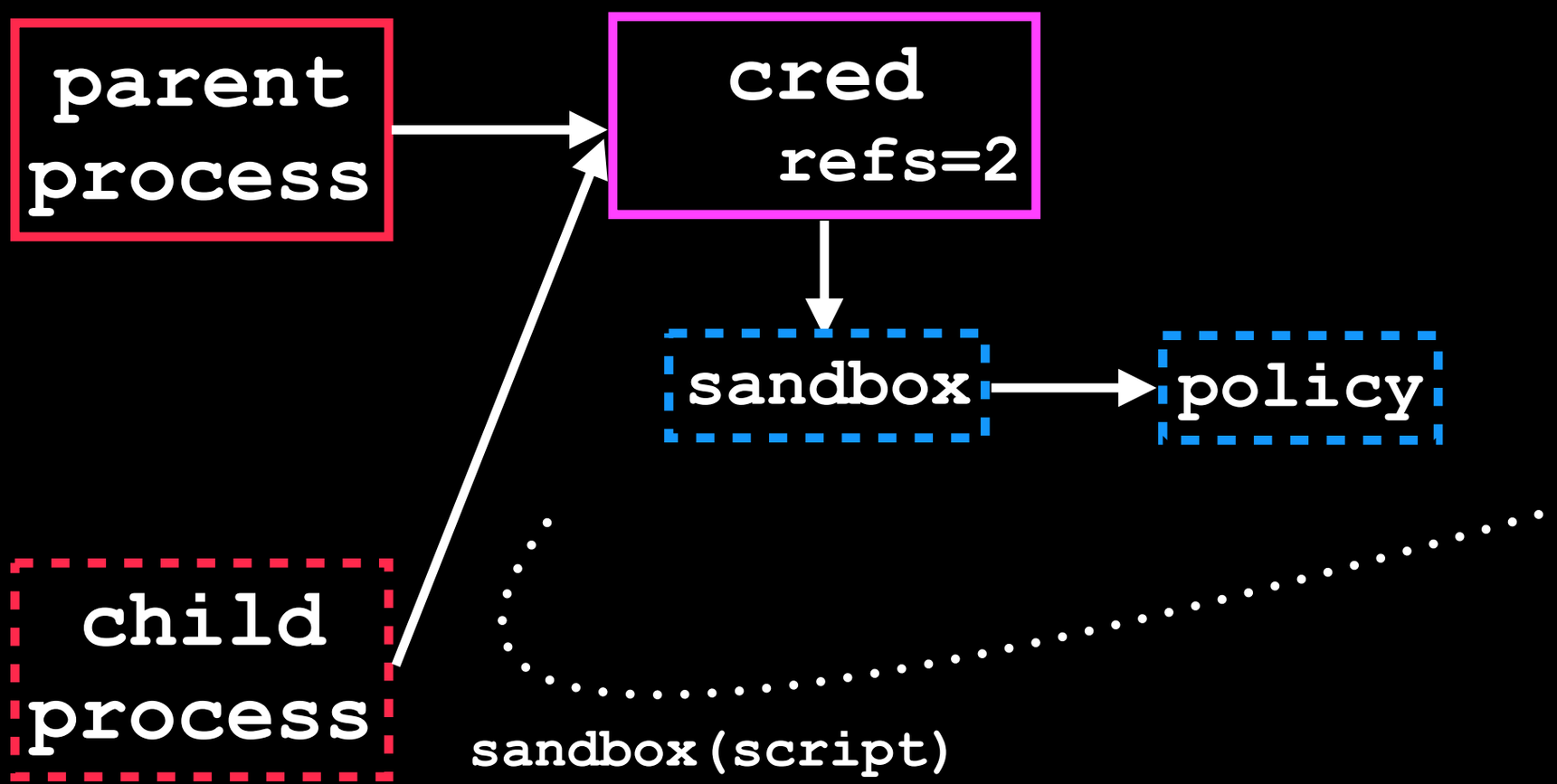
# process forking

**Normally, when the parent forks, the child process points to the same credential, and the credential's reference count is incremented.**



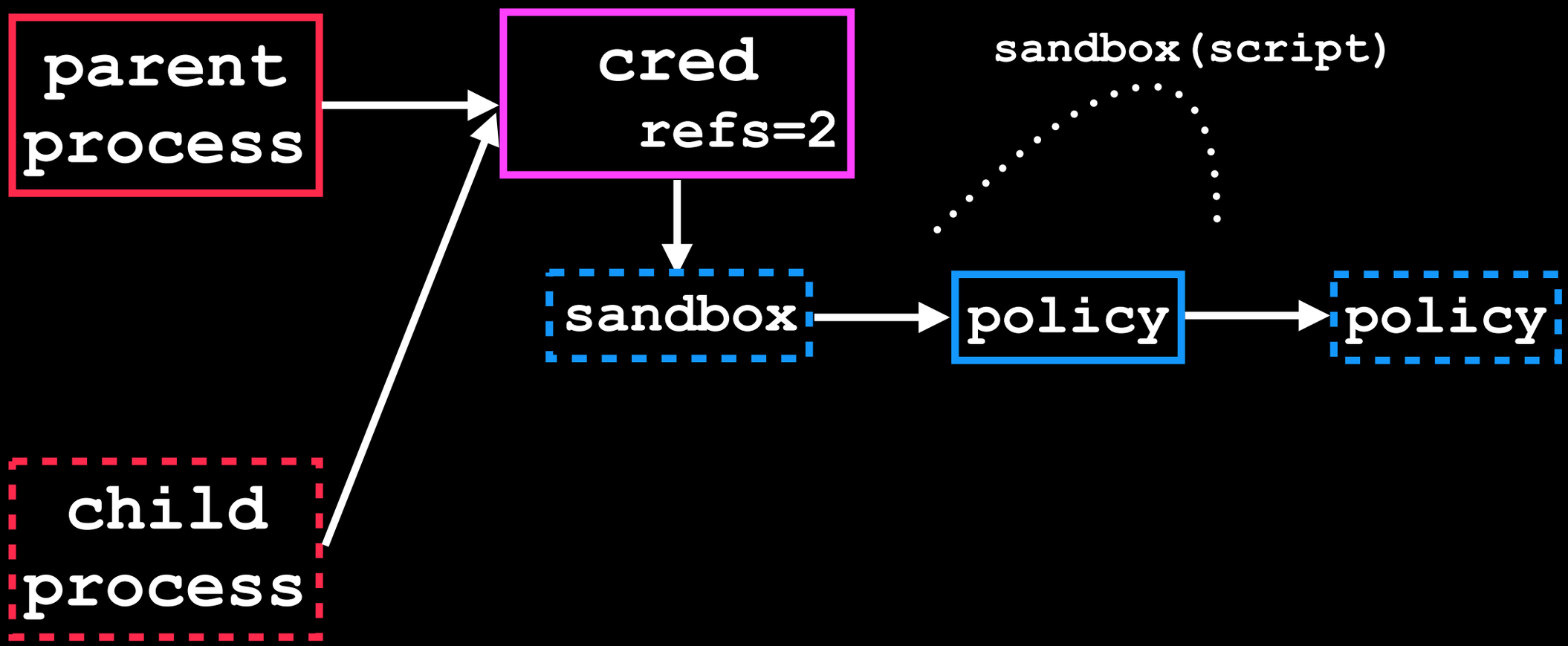
# process forking

**Normal forking behavior has the unfortunate consequence that if the child creates a sandbox, the sandbox is also applied to the parent.**



# process forking

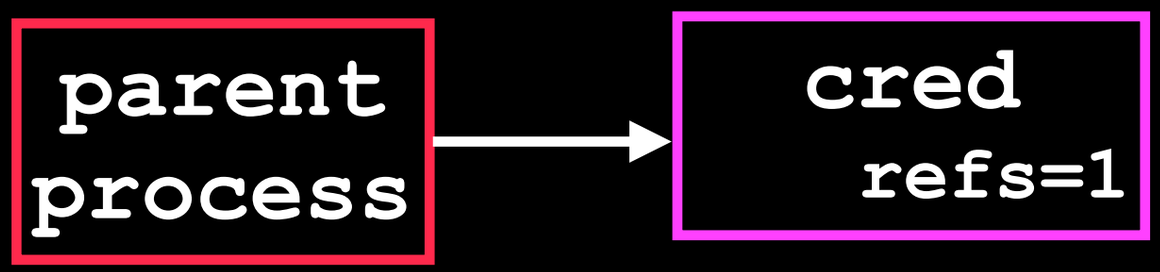
**Moreover, if the parent then adds a policy, the policy is also applied to the child.**

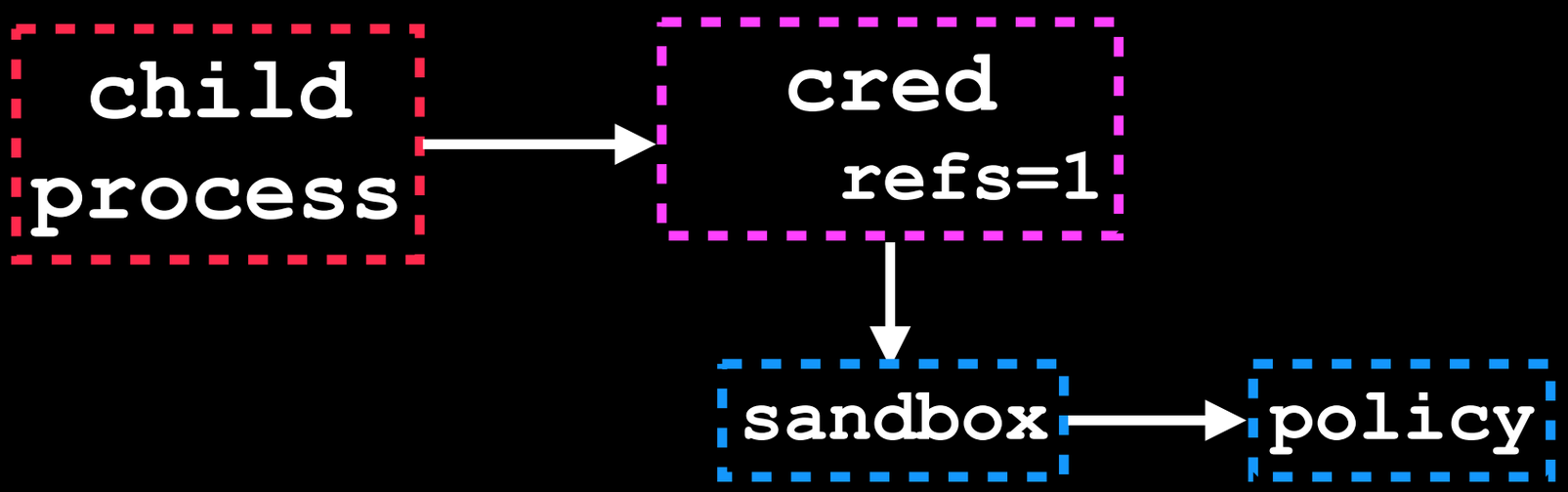


# process forking

**CASE 1: parent is not sandboxed and child creates a sandbox**

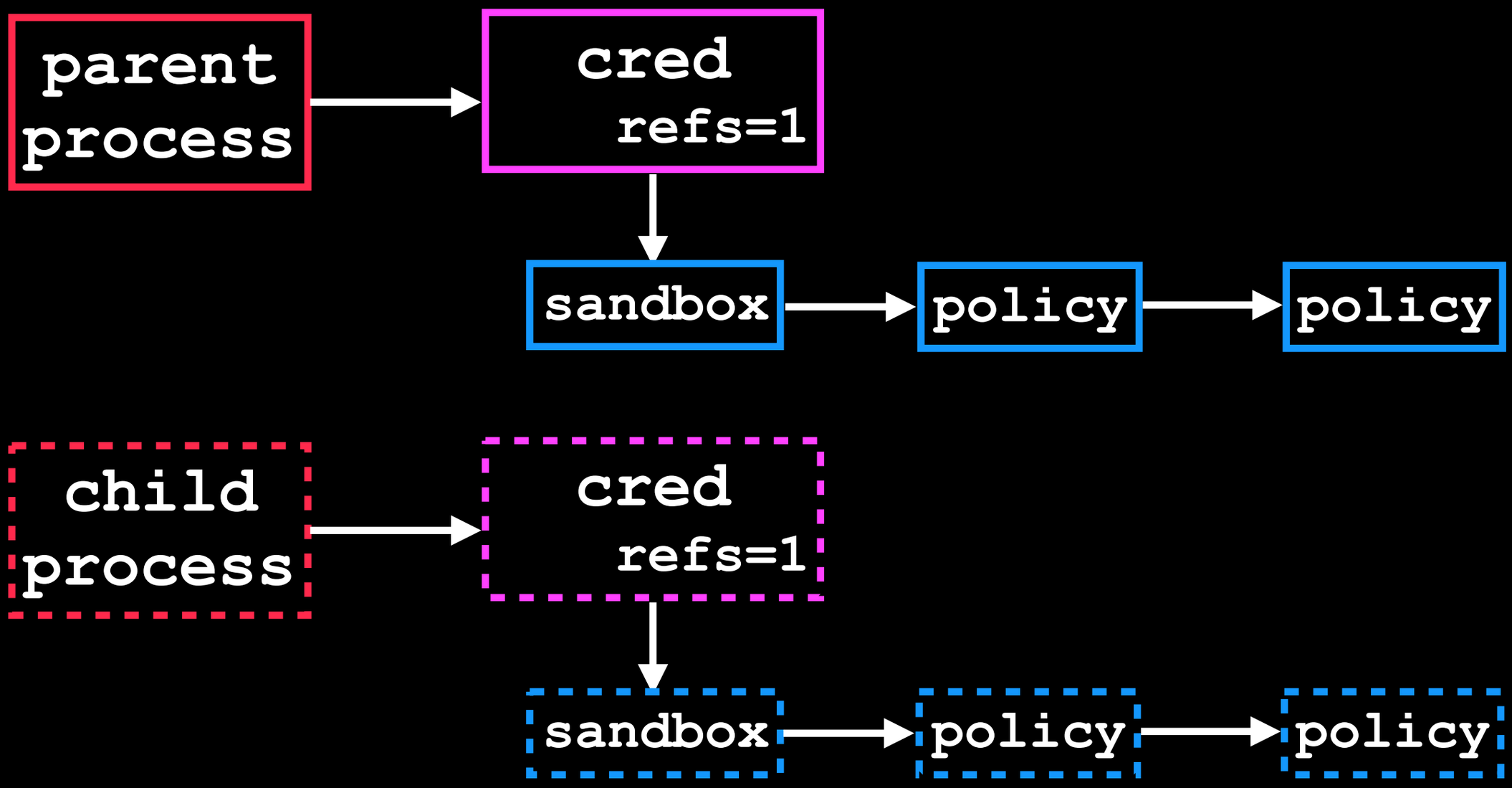
**secmodel_sandbox creates a new `cred` for the child when the child creates a sandbox.**



# process forking

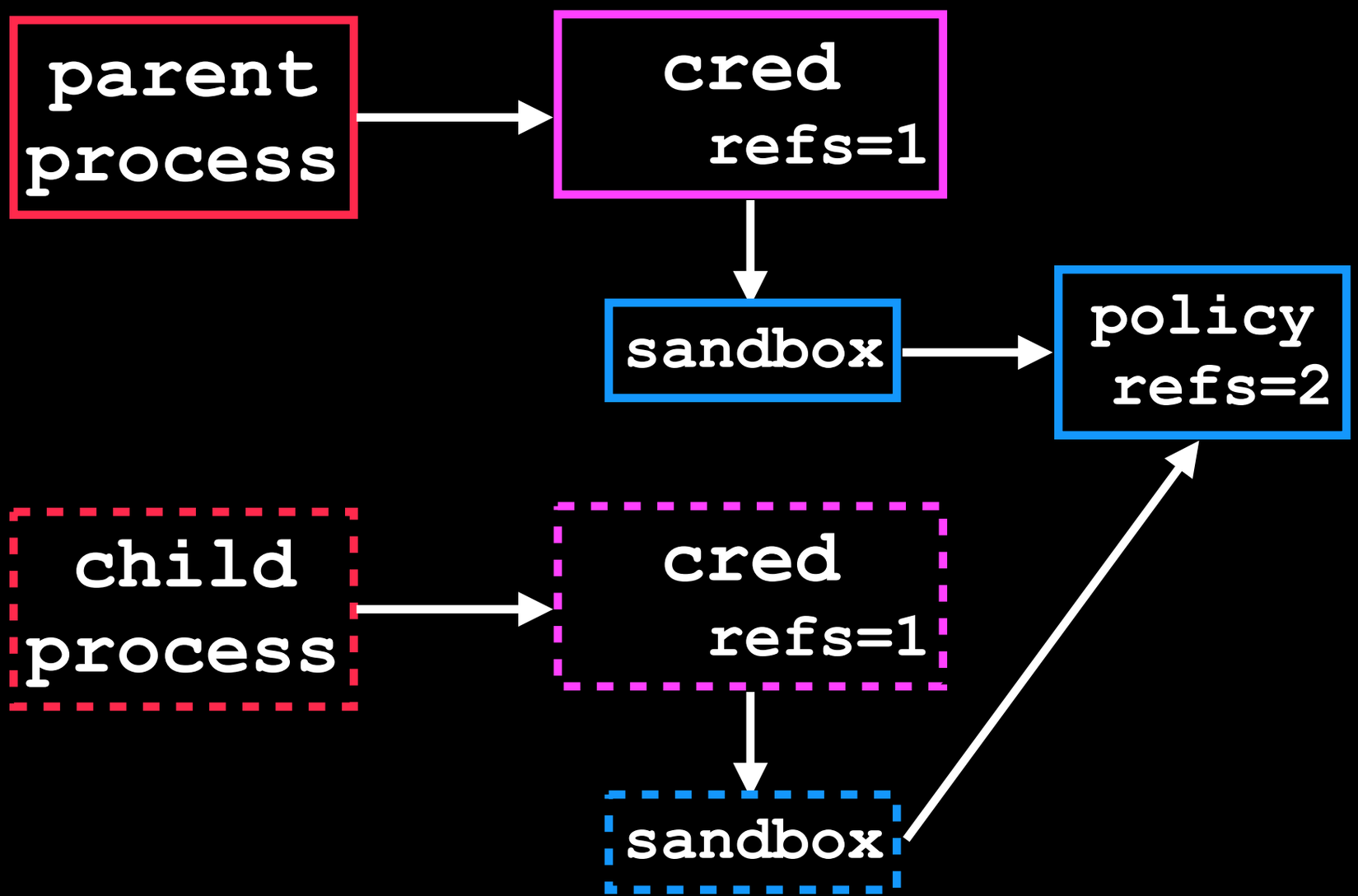
**Each process is then free to create its own sandboxes.**



# process forking

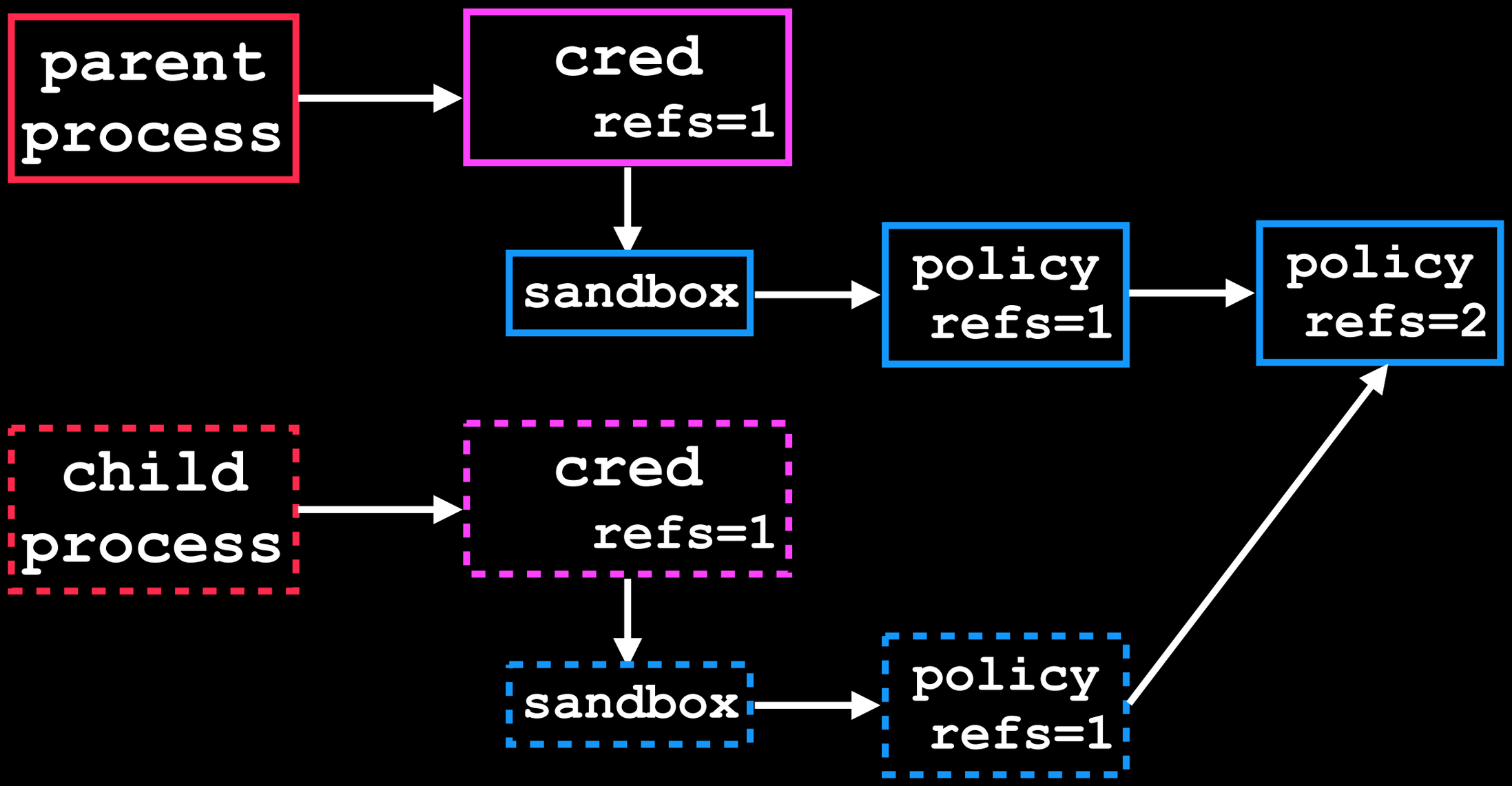
**CASE 2: parent already has a sandbox and forks**

**Child gets a new `cred` and a new `sandbox`. The child's sandbox points to the parent's newest policy. Policies are ref counted.**



# process forking

**Each process is free to further add its own policies.**



# process forking

Modification of the forking behavior uses kauth's cred scope, which notifies of events in a cred's lifecyle.

`fork` emits a `KAUTH_CRED_FORK` event. secmodel_sandbox handles this event by duplicating the parent's cred if the cred contains a sandbox.

Duplicating a cred emits a `KAUTH_CRED_DUP` event that secmodel_sandbox uses to create the `sandbox` in the child. The `sandbox`'s first member points to the most recent policy in the parent's cred.

# stateful policies

**Policy**

```
local _ = sandbox
local nsocks = 0

_.default('allow')
_.on('network.socket.open',
  function()
    nsocks = nsocks + 1
    if nsocks > 1 then
      return false
    else
      return true
    end
end)
```

**Program**

```
main()
{
  sandbox(POLICY);
  socket();

  /* any additional
   * calls to socket()
   * will fail
   */
}
```

# dynamic policies

**Policy**

```
local _ = sandbox;
_.default('allow')
_.deny('vnode')
_.on('process.signal',
  function(req, cred, p, sig)
    if sig == _.SIGUSR1 then
      _.allow('vnode')
      _.deny('network')
    end
    return true
  end)
```

**Program**

```
main()
{
  signal(SIGUSR1, noop);

  sandbox(POLICY);

  /* network, but not fs */
  data = wget();

  kill(getpid(),SIGUSR1);

  /* fs, but not network */
  read_file()
}
```

Inspired by: *Privacy Capsules: Preventing Information Leaks by Mobile Apps*. Raul Herbster, Scott DellaTorre, Peter Druschel, Bobby Bhattacharjee. *In Proceedings of the Fourteenth International Conference on Mobile Systems, Applications, and Services (MobiSys 2016)*. Singapore, June 2016.

# micro benchmarks

```
sandbox(POLICY)
for (i = 0; i < 10,000,000; i++) {
  syscall()
}
```

## sys time for 10,000,000 calls

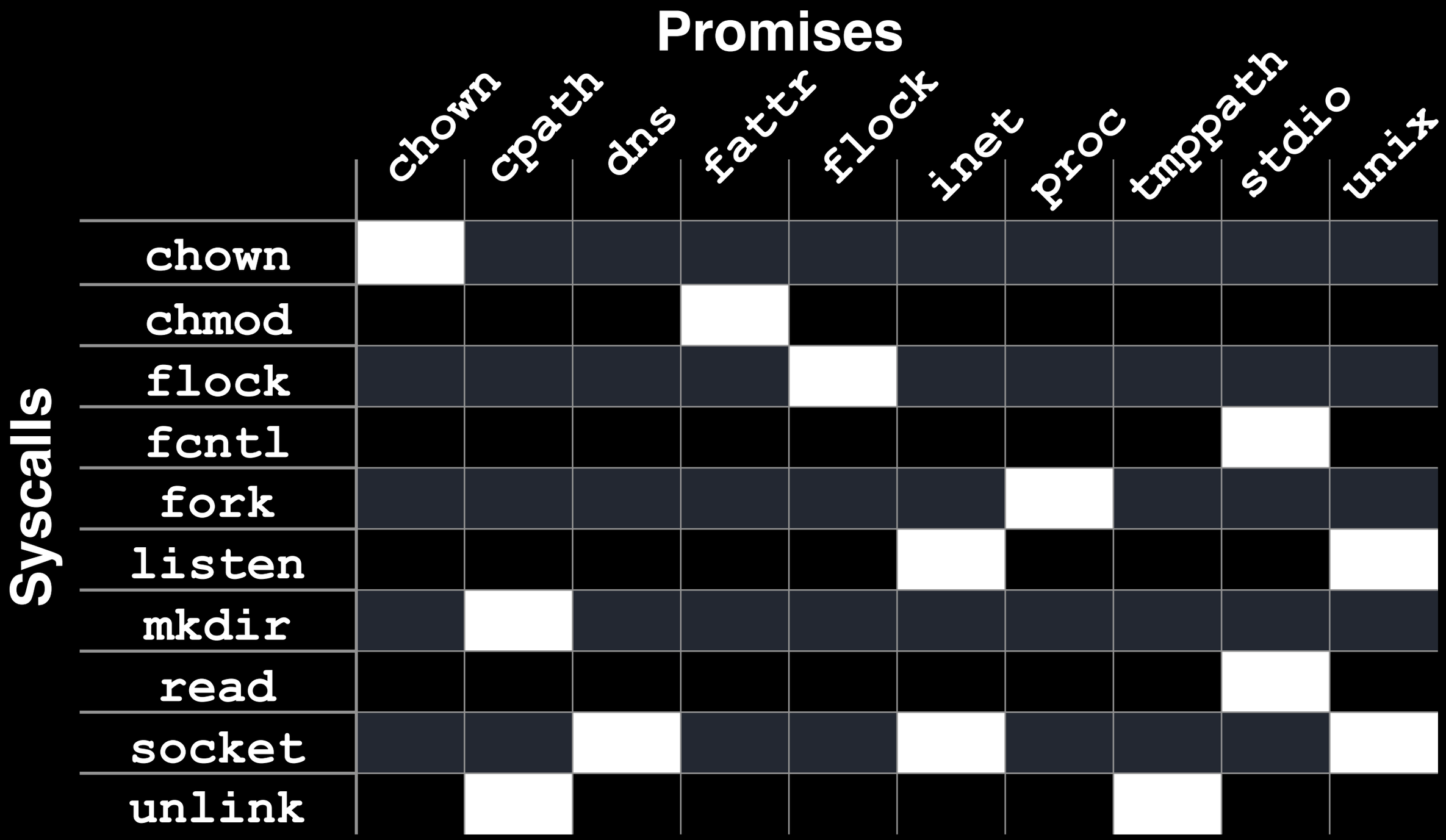
| | setpriority | socket |
|---|---|---|
| no sandbox | 1.597 | 14.725 |
| sandbox.allow() | 2.281 | 17.439 |
| sandbox.on() | 46.356 | 51.644 |

# OpenBSD's pledge

```
int pledge(const char *promises, const char *paths[])
```

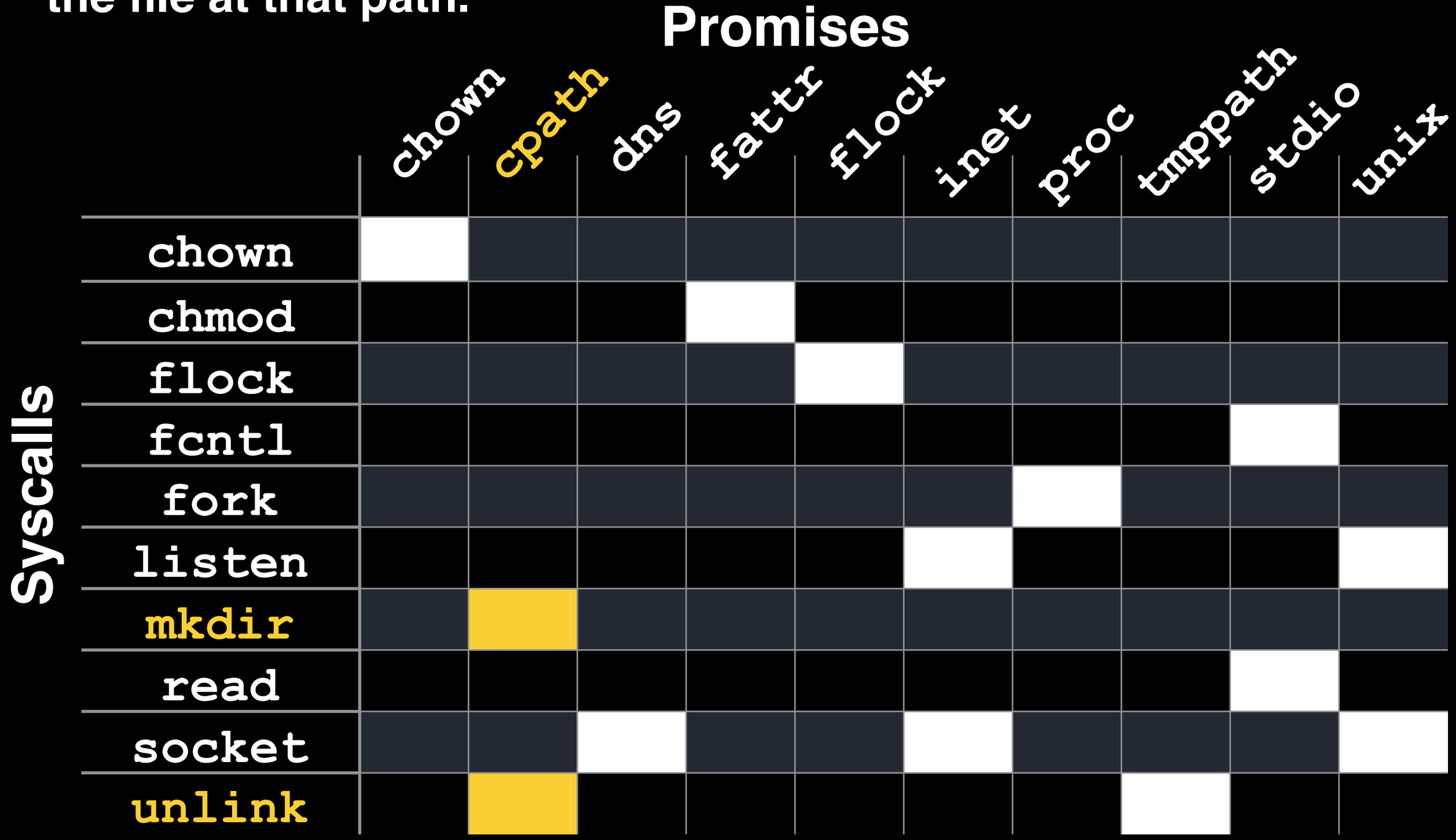
- POSIX syscalls grouped into categories
- restricts the process to the subset of POSIX as specified by the categories in `promises`
- If the process invokes a syscall outside of the promised subset, the process is killed

# OpenBSD's pledge



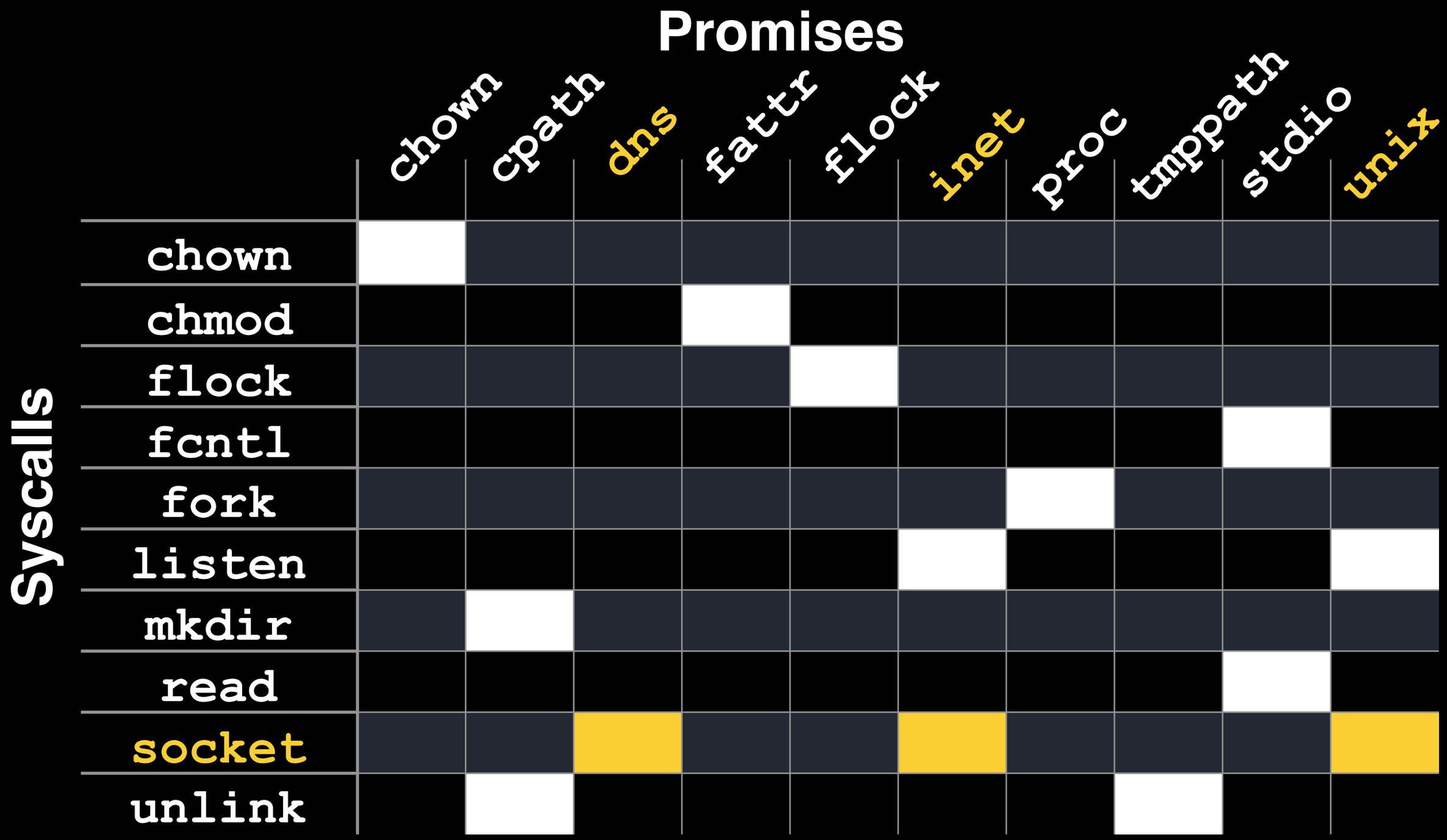
# OpenBSD's pledge

**`cpath`** allows syscalls taking a path argument that create or destroy the file at that path.



# OpenBSD's pledge

`socket` may be allowed if one of `dns`, `inet`, or `unix` is pledged.



# OpenBSD's pledge

When a syscall is trapped, the kernel checks:

Has the process called `pledge`?

YES. Has the process pledged any of the promises assigned to the syscall?

YES. Invoke the specific syscall handler.

NO. Kill the process.

Richer syscalls require additional argument/context checking.
Examples:

- `fcntl (stdio)`
  needs the `flock` promise if used for file locking.
- `unlink (cpath or tmppath)`
  If the file being deleted is outside of `/tmp`, then `cpath` is required.
- `socket (dns, inet, or unix)`
  The socket's domain must match a promise.

# emulating pledge with secmodel_sandbox

Ongoing effort.  Several challenges:

- kauth does not emit requests for many syscalls
  - memory-related functions, `setsockopt`, etc.

- slight but important platform differences
  - `sendsyslog`
  - `SOCK_DNS`

- semantic differences
  - secmodel_sandbox preserves sandbox across an exec, whereas pledge does not

# summary

**secmodel_sandbox is a new security model for NetBSD that allows per-process restriction of privileges.**

**Source code is available at:**

**www.cs.umd.edu/~smherwig/**